

Specyfikacje w praktyce na przykładzie JML-a

Sesja I

Co to jest JML?

- Java Modeling Language – Język Modelowania Javy
- Język wyrażania własności programów – logika pierwszego rzędu
- Składnia zrozumiała dla programistów
- "Design-by-contract" – projekt na bazie kontraktu
- `www.jmlspecs.org`

Co daje JML?

- Dokumentacja kodu
- Uproszczenie tworzenia programu przez abstrakcję
- Interfejs do rozmowy
- Rozwiązywanie konfliktów odpowiedzialności
- Rozwiązanie problemu niekompatybilności
- Sprawdzanie poprawności

Co daje JML? – dokumentacja

- Problem z `javadoc` – nieaktualne
- JML – dokładny w zakresie "co"
- JML – aktualny
- Mniejsze kłopoty z utrzymywaniem kodu

Co daje JML? – abstrakcja

- Człowiek pracuje wydajnie w określonych ramach
- Za ciężkie zadanie – da się wykonać, ale z błędami
- Specyfikacja – wydzielenie części ciężaru
- Specyfikacja mówi "co", stara się nie mówić "jak"

Co daje JML? – interfejs, odpowiedzialność

- Zleceniodawca wykonania programu mówi "co" chce
- Zleceniobiorca po wykonaniu może powiedzieć: to jest moja realizacja "co"
- Zmniejszenie pola do oskarżeń/odwołań itp.
- Jasne jest, gdzie leży wina

Co daje JML? – kompatybilność

- Różne implementacje tej samej biblioteki
- Problemy ze zgodnością co do podstawowej funkcjonalności
- Precyzyjny opis – brak niezgodności

Co daje JML? – sprawdzanie

- Możemy oczekiwać od kodu oczekiwanej własności
- Zapewnienie, że ta własność zachodzi

Co daje JML? – sprawdzanie, narzędzia

- Kompilacja z asercjami sprawdzanymi w czasie wykonania – *jmlc (jmlrac)*
- Przekształcanie specyfikacji w testy unitarne – *jmlunit*
- Sprawdzanie statyczne specyfikacji – *ESC/Java2*
- Weryfikacja programów – *LOOP, JACK, KeY, Krakatoa*
- Narzędzia wspomagające – *Houdini, Daikon, Canapa, jmlspec*

Co to jest wadą JML?

- Pracochłonność (ukryty koszt staje się jawny)
(klienta nie obchodzi czy młotek ma równe krawędzie)
- Narzędzia trudne w obsłudze
- Często specyfikacja == implementacja
- W niektórych zastosowaniach – nie do użycia

Dodatkowe zadanie domowe

- Napisać specyfikacje w JML-u do wskazanej klasy MIDP
- Procedura: zgłosić się do mnie jak najszybciej
- Terminarz:
 - do 20.01 wysłać specyfikacje
 - do około 25.01 sprawdzone (na umowienie)
- Zysk:
Zaliczenie pracy domowej =>
zaliczenie zadania z weryfikacji na egzaminie przy
niezmienionym czasie trwania egzaminu

JML – warunki wejścia i wyjścia

- `requires`, `ensures`
- Można opisywać za pomocą wyrażeń warunki wejścia i wyjścia metod

```
/*@ requires amount >= 0;
    ensures balance == \old(balance-amount) &&
           \result == balance;

@*/
public int debit(int amount) {
    ...
}
```

JML – warunki wejścia i wyjścia c.d.

- Specyfikacje w JML-u mogą być dowolnie silne/słabe

```
/*@ requires amount >= 0;  
   ensures true;  
  @*/  
public int debit(int amount) {  
    ...  
}
```

Domyślny warunek wyjścia – `true` można opuścić.

JML – warunki wejścia i wyjścia c.d.

- Można specyfikować wiele par requires-ensures:

```
/*@   requires amount >= 0;
      ensures true;
      also
      requires amount < 0 && webInterface;
      ensures true;

  @*/
public int debit(int amount) {
    ...
}
```

- Semantyka:
 - metodę można wywołać bezkarnie, gdy dowolne z **requires** jest spełnione,
 - jeśli dane **requires** jest spełnione na wejściu, to spełnione jest też odpowiednie **ensures**

Wyjątki w specyfikacjach – dwie formy

- Na poprzednich slajdach było domyślnie `normal_behavior`
- Uwaga na różnicę między
 1. jeśli zachodzi P , to wyrzucone ma być `SomeException`
 2. jeśli `SomeException` jest wyrzucone, to zachodzi P
- Łatwo te rzeczy pomylić
- Wyrażanie
 1. za pomocą `exceptional_behavior`
 2. za pomocą `signals`

Wyjątki w specyfikacjach –

exceptional_behavior

- Przykład użycia

```
/*@ exceptional_behavior
    requires amount > balance;
    signals (BankException e)
        e.getReason.equals("Amount too big");
@*/
public int debit(int amount) {
    ...
}
```

mówi, że `BankException` musi być wyrzucone, gdy `amount > balance`.

Wyjątki w specyfikacjach – `signals`

- Przykład użycia

```
/*@ exceptional_behavior
    requires amount <= balance;
    signals (BankException e)
        e.getReason.equals("Database error");
@*/
public int debit(int amount) {
    ...
}
```

mówi, że `BankException` może być wyrzucone, gdy `amount <= balance`, ale powód jest "nielogiczny".

Wyjątki w specyfikacjach – implicite

- Domyślnie metoda może rzucać wyjątki, ale tylko te z klauzuli `throws`, zatem

```
//@ requires 0 <= amount && amount <= balance;  
public int debit(int amount)  
           throws BankException { ... }
```

ma implicite klauzulę:

```
signals (BankException) true;
```

oraz klauzulę:

```
signals (Exception e) e instanceof BankException;
```

Wyjątki w specyfikacjach – implicite

- Domyślnie metoda może rzucać wyjątki, ale tylko te z klauzuli `throws`, zatem

```
//@ requires 0 <= amount && amount <= balance;  
public int debit(int amount) {  
    ...  
}
```

ma domyślnie klauzulę

```
signals (Exception) false;
```

Przy okazji – `debit` nie może wyrzucić także nieraportowanego wyjątku, choć Java nie wymaga dla takich wyjątków raportu w `throws`

Efekty uboczne – assignable

- Własności ramek ograniczają możliwe efekty uboczne metod:

```
/*@
    requires amount >= 0;
    assignable balance;
    ensures balance == \old(balance)-amount;
@*/
public int debit(int amount) {
    ...
}
```

- To oznacza, że metoda `debit` może przypisywać tylko do pola `balance`.
- To nie wynika z warunku wyjścia
- Domyślna klauzula: `assignable \everything`

Efekty uboczne – pure

- Metoda bez efektów ubocznych jest określana jako **pure**

```
public /*@ pure @*/ int getBalance() {...}
```

```
Directory /*@ pure non_null @*/ getParent() {...}
```

- Metody **pure** mają domyślnie assignable \nothing.
- W specyfikacjach można używać (wyłącznie) metod **pure**, np.:

```
/*@ invariant 0<=getBalance() &&  
           getBalance() <=MAX_BALANCE;  
   @*/
```

JML – niezmienniki

- Niezmienniki klasowe, w odróżnieniu od niezmienników pętli
- Muszą być utrzymywane przez wszystkie metody, np.

```
public class Wallet {  
    public static final short MAX_BAL = 1000;  
    private short balance;  
    /*@ invariant 0 <= balance &&  
                balance <= MAX_BAL;  
    @*/  
    ...  
}
```

- Niezmienniki są niejawnie dołączane do wszystkich warunków wstępnych i warunków końcowych

JML – niezmienniki c.d.

- Niezmienniki muszą być zachowywane także po wyrzuceniu wyjątku!
- Niezmienniki innych obiektów – zakładane przed wejściem do metody

JML – niezmienniki c.d.

- Niezmienniki dokumentują decyzje projektowe, np.:

```
public class Directory {
    private File[] files;
    /*@ invariant files != null &&
        (\forall int i; 0 <= i && i < files.length;
            files[i] != null &&
            files[i].getParent() == this)
    @*/
```

- Jawne zapisywanie niezmienników pomaga w zrozumieniu kodu.

JML – niezmienniki c.d.

- Rodzajem niezmiennika są niezmienniki temporalne
`constraint`

- Opisują ewolucję stanu, np.:

```
public class Gadgets {  
    int counter = 0;  
    /*@ constraint counter >= \old(counter);  
    @*/
```

- Semantyka: relacja między wartościami przed wywołaniem dowolnej metody i po jej wywołaniu

Niezmienniki a pre- i post-warunki

- Niezmienniki i klauzule `constraint` generują dodatkowe warunki końcowe dla każdej z metod

```
public class A {  
    int counter = 0;  
    /*@ invariant Q; @*/  
    /*@ constraint P; @*/  
    void m() { ...  
    }
```

Niezmienniki a pre- i post-warunki

- Niezmienniki i klauzule `constraint` generują dodatkowe warunki końcowe dla każdej z metod

```
public class A {  
    int counter = 0;  
    /*@ requires true;  
       ensures P && Q;  
    @*/  
    void m() { ...  
    }
```

Specyfikacje behawioralne

1. Typ klasy: konstruktory, metody (z sygnaturami)
2. To nie opisuje zachowania
3. W JML-u typ to: jak w Javie + opis zachowania

Uszczegółowienie

- Uszczegółowienie C specyfikacji A to taki opis, że każda implementacja C jest także implementacją A .
- Tłumacząc na prewarunki i postwarunki:
Oznaczmy przez R_A, R_C prewarunki, zaś przez E_A, E_C odpowiednie postwarunki, powinno zachodzić:
 - uszczegółowienie powinno mieć tę samą składnię (nazwy metod, liczby argumentów itp.)
 - prewarunki są związane: $R_C \Rightarrow R_A$
 - postwarunki są związane: $(R_C \Rightarrow E_C) \Rightarrow (R_A \Rightarrow E_A)$

Behawioralny podtyp

- Podklasa – uszczegółowienie typu klasy
- Podsumowanie
 - relacja bycia podklasą – gwarantuje poprawność strukturalną (obecność pól, metod itp.)
 - relacja bycia podtypem – gwarantuje brak błędów typowych, gdy podtypy są używane w miejscu typów
 - relacja bycia podtypem behawioralnym – gwarantuje brak problemów z dziwnym zachowaniem, gdy podtypy są używane w miejscu typów

Behawioralny podtyp w JML-u

- Nadpisywana metoda dziedziczy specyfikacje z nadklasy
- Można dodać bardziej szczegółowy opis za pomocą słowa kluczowego `also`

Behawioralny podtyp w JML-u – niezmienniki

- Niezmienniki są dziedziczone do podklas:

```
class Parent {  
    ...  
    //@ invariant invParent;  
    ...  
}  
class Child extends Parent {  
    ...  
    //@ invariant invChild;  
    ...  
}
```

niezmiennik w klasie Child to `invChild && invParent`

Behawioralny podtyp w JML-u – metody

- W wypadku specyfikacji metod wygląda to tak:

```
class Parent {
    //@ requires i >= 0;
    //@ ensures \result >= i;
    int m(int i){ ...
}

class Child extends Parent {
    //@ also
    //@ requires i <= 0
    //@ ensures \result <= i;
    int m(int i){ ...
}
}
```

Słowo kluczowe `also` wskazuje, że specyfikacje są dziedziczone.

Behawioralny podtyp w JML-u – metody c.d.

- Metoda `m` w `Child` musi przestrzegać obu specyfikacji, więc pełna specyfikacja dla niej to:

```
class Child extends Parent {
    /*@ requires i >= 0;
       @ ensures \result >= i;
       @ also
       @ requires i <= 0;
       @ ensures \result <= i;
    @*/
    int m(int i){ ... }
}
```

Jakich wyników możemy się spodziewać?

Behawioralny podtyp w JML-u – metody c.d.

- Jeszcze inny sposób przedstawienia tej specyfikacji:

```
class Child extends Parent {
    /*@ requires i >= 0 || i <= 0;
       @ ensures \old(i)>=0 ==> \result >= i;
       @ ensures \old(i)<=0 ==> \result <= i;
    @*/
    int m(int i){ ... }
}
```

Behawioralny podtyp – różne

- Można „osłabić” behawioralność
- Należy zrelatywizować typ:
`\typeof(this) == \type(Nadklasa)`
- Dziedziczone też jest: `normal_behavior,`
`exceptional_behavior,`
`signals (E e) false in.`

Za tydzień

- Jeszcze trochę teorii
- Przykład wyspecyfikowanej klasy
- Co się może stać, gdy człowiek sobie obspecyfikuje?